Chapter TWENTY-SEVEN
Concurrency

Exam Objectives

Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution.
Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList.

Synchronization

In the last chapter, we reviewed some problems that can happen in a concurrent environment and briefly talked about locking.

For example, one solution to avoid a race condition is to ensure that only one thread at a time can access the code that causes the problem; a process known as *synchronizing* that block of code.

Synchronization works with locks. Every object comes with a built-in lock and since there is only lock per object, only one thread can hold that lock at any time. The other threads cannot take the lock until the first thread releases it. Meanwhile, they are blocked.

You define a lock by using the `synchronized` keyword on either a block or a method. That lock is acquired when a thread enters an unoccupied synchronized block or method.

In a synchronized block, you use the `synchronized` keyword followed by either a reference variable:

```
Object o = new Object();

synchronized (o) { // Get the lock of Object o
    // Code guarded by the lock
}
```

Or the `this` keyword:

```
// Get the lock of the object this code belongs to
synchronized (this) {
```

```
    // Code guarded by the lock
}
```

The lock is released when the block ends.

You can also synchronize an entire method:

```
public synchronize void method() {
    // Code guarded by the lock
}
```

In this case, the lock belongs to the object on which the method is declared and is released when the method ends.

Notice that synchronizing a method is equivalent to this:

```
public void method() {
    synchronized (this) {
        // Code guarded by the lock
    }
}
```

Static code can also be synchronized but instead of using `this` to acquire the lock of an instance of the class, it is acquired on the single class object that every instance of the class has associated:

```
class MyClass {
    synchronized static void method() {
        /** .. */
    }
}
```

Is equivalent to:

```
class MyClass {
    static void method() {
        synchronized (MyClass.class) {
            /** ... */
        }
    }
}
```

Now, for example, if we execute the following code:

```
public class Test {
    static int n = 0;
    static void add() {
        n++;
        System.out.println(n);
    }
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(4);
        Runnable r = () -> add();
        for(int i = 0; i < 4; i++) {
            service.execute(r);
        }
        service.shutdown();
    }
}
```

By looking at one possible output, we can see that we have a race condition problem because four threads are executing the same code:

```
2
2
3
4
```

But when we synchronize the `add()` method to ensure only one thread can access it, the problem goes away:

```
1
2
3
4
```

Atomic Classes

The `java.concurrent.atomic` package contains classes to perform atomic operations, which are performed in a single step without the intervention of more than thread.

We have for example:

`AtomicBoolean` , `AtomicInteger` , `AtomicLong` , and `AtomicReference<V>`
To update a value of the corresponding type (or object reference) atomically.

`AtomicIntegerArray` , `AtomicLongArray` , and `AtomicReferenceArray<E>`
To update the elements of the corresponding array type (or object reference) atomically.

Each class has methods that perform operations like increments or updates in an atomic way. Take for example the `AtomicInteger` class (the other classes have similar methods):

`int addAndGet(int delta)`
Atomically adds the given value to the current value.

`boolean compareAndSet(int expect, int update )`
Atomically sets the value to the given updated value if the current value equals the expected value.

`int decrementAndGet()`
Atomically decrements by one the current value.

`int get()`
Gets the current value.

`int getAndAdd(int delta)`
Atomically adds the given value to the current value.

`int getAndDecrement()`
Atomically decrements by one the current value.

`int getAndIncrement()`
Atomically increments by one the current value.

`int getAndSet(int newValue)`
Atomically sets to the given value and returns the old value.

`int incrementAndGet()`
Atomically increments by one the current value.

`void set(int newValue)`
Sets to the given value.

The example about incrementing a variable we reviewed before can be rewritten as:

```
public class Test {
    static AtomicInteger n = new AtomicInteger(0);
    static void add() {
        System.out.println(n.incrementAndGet());
    }
    public static void main(String[] args) {
        ...
    }
}
```

When we execute this, we'll get all the numbers but not in order, because atomic classes only ensure data consistency:

```
1
4
2
3
```

Concurrent Collections

The `java.util.concurrent` package provides some thread-safe classes equivalent to the collection classes of `java.util` .

This way, instead of explicitly synchronizing an operation like this:

```
Map<String, Integer> map = new HashMap<>();
...
void putIfNew(String key, Integer val) {
    if(map.get(key) == null) {
        map.put(key, val);
    }
}
```

You can use a `ConcurrentHashMap` like this:

```
Map<String, Integer> map = new ConcurrentHashMap<>();
...
map.putIfAbsent(key, val);
```

So when working in concurrent environments, if you're to modify collections, it's better to use the collections of `java.util.concurrent` (besides, they often perform better than plain synchronization).

In fact, if you only need plain `get()` and `put()` methods (when working with a map, for example) you only have to change the implementation:

```
//Map<String, Integer> map = new HashMap<>();
Map<String, Integer> map = new ConcurrentHashMap<>();
map.put("one", 1);
Integer val = map.get("one");
```

Java provides a lot of concurrent collections, but we can classify them by the interface they implement:

- `BlockingQueue` (for queues)
- `BlockingDeque` (for deques)
- `ConcurrentMap` (for maps)
- `ConcurrentNavigableMap` (for navigable maps like `TreeMap`)
- `CopyOnWriteArrayList` (for lists)

Since these interfaces extends from the `java.util` collection interfaces, they inherit the methods we already know (and behave like one of them) so here you'll only find the added methods that support concurrency.

BlockingQueue
It represents a thread-safe queue that waits (with an optional timeout) for an element to be inserted if the queue is empty or for an element to be removed if the queue is full:

	Blocks	Times Out
Insert	`void put(E e)`	`boolean offer(E e, long timeout, TimeUnit unit)`
Remove	`E take()`	`E poll(long timeout, TimeUnit unit)`

After the waiting time elapses, `offer()` returns `false` and `poll()` returns `null` .

The main implementations of this interface are:

- `ArrayBlockingQueue` . A bounded blocking queue backed by an array.
- `DelayQueue` . An unbounded blocking queue of `Delayed` elements, in which an element can only be taken when its delay has expired.
- `LinkedBlockingQueue` . An optionally-bounded blocking queue based on linked nodes.
- `LinkedTransferQueue` . An unbounded `TransferQueue` based on linked nodes.
- `PriorityBlockingQueue` . An unbounded blocking queue that uses the same ordering rules as class `PriorityQueue` and supplies blocking and retrieval operations.
- `SynchronousQueue` . A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. You cannot peek at a synchronous queue because an element is only present when you try to remove it.

BlockingDeque
It represents a thread-safe deque (a double-ended queue, a queue which you can insert and take elements from in both ends). It extends from `BlockingQueue` and `Deque` , and also waits (with an optional timeout, after which, it returns `false` or `null` too) for an element to be inserted if the deque is empty or for an element to be removed if the deque is full:

First Element (head)		
	Blocks	**Times Out**
Insert	`void putFirst(E e)`	`boolean offerFirst(E e, long timeout, TimeUnit unit)`
Remove	`E takeFirst()`	`E pollFirst(long timeout, TimeUnit unit)`
Last Element (tail)		
	Blocks	**Times Out**
Insert	`void putLast(E e)`	`boolean offerLast(E e, long timeout, TimeUnit unit)`
Remove	`E takeLast()`	`E pollLast(long timeout, TimeUnit unit)`

ConcurrentMap

This interface represents a thread-safe map and it's implemented by the `ConcurrentHashMap` class.

Here are some of its most important methods:

`V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
Atomically computes the value of a specified key with a `BiFunction`

`V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`
Atomically computes the value of a key only if it's not present in the map

`V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`
Atomically computes the value of a key only if it's present in the map

`V getOrDefault(Object key, V defaultValue)`
Returns the value of the key or a default value if the key is not present

`V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)`
If the specified key is not already associated with a (non-`null`) value, associates it with the given value. Otherwise, replaces the value with the results of the given remapping function, or removes if `null`. This is performed atomically.

`V putIfAbsent(K key, V value)`
If the key is not present in the map, it's put with the given value atomically

ConcurrentNavigableMap

It represents a thread-safe `NavigableMap` (like `TreeMap`) and is implemented by the `ConcurrentSkipListMap` class, sorted according to the natural ordering of its keys, or by a `Comparator` provided in its constructor.

A `ConcurrentNavigableMap` also implements `Map` and `ConcurrentMap`, so it has methods like `computeIfAbsent()`, `getOrDefault()`, etc.

CopyOnWriteArrayList

It represents a thread-safe `List`, similar to an `ArrayList`, but when it's modified (with methods like `add()` or `set()`), a new copy of the underlying array is created (hence the name).

When iterating an `ArrayList`, methods like `remove()`, `add()` or `set()` can throw a `java.util.ConcurrentModificationException`. With a `CopyOnWriteArrayList`, this exception is not thrown because the iterator works with an unmodified copy of the list. But this also means that calling, for example, the `remove()` method on the `Iterator` is not supported (it throws an `UnsupportedOperationException`).

Consider this example, where the last element of the list is changed in every iteration but the original value is still printed inside the iterator:

```
List<Integer> list = new CopyOnWriteArrayList<>();
list.add(10); list.add(20); list.add(30);
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
    int i = it.next();
    System.out.print(i + " ");
    // No exception thrown
    list.set(list.size() -1, i * 10);
    // it.remove(); throws an exception
}
System.out.println(list);
```

The output:

```
10 20 30 [10, 20, 300]
```

With a `CopyOnWriteArrayList`, there is no lock on reads, so this operation is faster. For that reason, `CopyOnWriteArrayList` is most useful when you have few updates and inserts and many concurrent reads.

There are other classes like `ConcurrentSkipListSet`, which represents a thread-safe `NavigableSet` (like a `TreeSet`).

Or `CopyOnWriteArraySet`, which represents a thread-safe `Set` and internally uses a `CopyOnWriteArrayList` object for all of its operations (so these classes are very similar).

Besides all these classes, for each type of collection, the `Collections` class has methods like the following that take a normal collection and wrap it in a synchronized one:

```
static <T> Collection<T> synchronizedCollection(Collection<T> c)
static <T> List<T> synchronizedList(List<T> list)
static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
static <T> Set<T> synchronizedSet(Set<T> s)
```

However, it's required to synchronize these collections manually when traversing them via an `Iterator` or `Stream`:

```
Collection c =
   Collections.synchronizedCollection(myCollection);
...
synchronized (c) {
   Iterator i = c.iterator();
   ...
}
```

Also, they throw an exception if they are modified within an iteration.

Only use these methods if you have to work with an existing collection in a concurrent environment (otherwise, from the beginning, use a `java.util.concurrent` collection).

CyclicBarrier

The `synchronized` keyword helps us coordinate access to a shared resource by multiple threads.

But this is very low-level work. I mean, it takes a lot of effort to coordinate complex concurrent tasks. Luckily, Java provides high-level classes to more easily implement these kinds of synchronization tasks.

One of these classes is `CyclicBarrier`. It provides a synchronization point (a barrier point) where a thread may need to wait until all other threads also reach that point.

This class has two constructors:

`CyclicBarrier(int threads)`
Creates a `CyclicBarrier` with the specified number of threads waiting for it.

`CyclicBarrier(int parties, Runnable barrierAction)`
Creates a `CyclicBarrier` with the specified number of threads waiting upon it, and

which will execute the given action when the barrier is reached.

The methods:

```
int await()
   throws InterruptedException, BrokenBarrierException
int await(long timeout, TimeUnit unit)
   throws InterruptedException,
          BrokenBarrierException,
          TimeoutException
```

Block a thread until all the other threads have called `await()` (reached the barrier), or optionally, until the specified waiting time elapses (when this happens, a `TimeoutException` is thrown).

These methods throw an `InterruptedException` if the current thread was interrupted while waiting and a `BrokenBarrierException` if another thread was interrupted or timed out, or the barrier was reset (with the `reset()` method), or the barrier action failed due to an exception.

Here's an example:

```
public class CyclicBarrierExample {
   static void checkStep(
            CyclicBarrier barrier, String step) {
      // Do something to prepare the step
      System.out.println(step + " is ready");
      try {
         // Wait the other threads
         barrier.await();
      } catch (Exception e) { /** ... */ }
   }
   public static void main(String[] args) {
      String[] steps = {"Read the recipe",
         "Gather the ingredients",
         "Wash hands"};
      System.out.println("Preparing everything:");

      Runnable allSet = () ->
         System.out.println( "Everything's ready. Let's begin.");

      ExecutorService executor =
         Executors.newFixedThreadPool(steps.length);
      CyclicBarrier barrier =
         new CyclicBarrier(steps.length, allSet);

      for(String step : steps) {
         executor.submit(
            () -> checkStep(barrier, step)
         );
      }

      executor.shutdown();
   }
}
```

The output:

```
Preparing everything:
Gather the ingredients is ready
Read the recipe is ready
Wash hands is ready
Everything's ready. Let's begin.
```

We have three steps and one thread to process each one. The threads will print the given step and call the `await()` method. When the three threads have called that method, the `Runnable` represented by the `allSet` variable is executed (the one that prints the `"Everything's ready..."` message).

As you can see, the steps are not printed in order, but the program cannot proceed until all of them are executed.

Notice that the `CyclicBarrier` is created with the same number of threads. It has to be this way. Otherwise, the program will wait forever.

If the number of threads is less than the `CyclicBarrier` expects, it will wait forever for the missing threads.

To understand why the program will block when the number of threads is greater, you need to know that a `CyclicBarrier` can be reused and how that works.

When all the expected threads call `await()`, the number of waiting threads on the `CyclicBarrier` goes back to zero and it can be used again for a new set of threads. This way, if the `CyclicBarrier` expects three threads but there are four, a cycle of three will be completed and for the next, two will be missing, which will block the program.

Key Points

- Synchronization works with locks. Every object comes with a built-in lock and since there is only lock per object, only one thread can hold that lock at any time. You acquire a lock by using the `synchronized` keyword in either a block or a method.
- Static code can also be synchronized but instead of using this to acquire the lock of an instance of the class, it is acquired on the class object that every class has associated.
- The `java.concurrent.atomic` package contains classes (like `AtomicInteger`) to perform atomic operations, which are performed in a single step without the intervention of more than thread.
- `BlockingQueue` represents a thread-safe queue and `BlockingDeque` represents a thread-safe deque. Both wait (with an optional timeout) for an element to be inserted if the queue/deque is empty or for an element to be removed if the queue/deque is full.
- `ConcurrentMap` represents a thread-safe map and it's implemented by the `ConcurrentHashMap` class. `ConcurrentNavigableMap` represents a thread-safe `NavigableMap` (like `TreeMap`) and is implemented by the `ConcurrentSkipListMap` class.
- `CopyOnWriteArrayList` represents a thread-safe `List`, similar to an `ArrayList`, but when it's modified (with methods like `add()` or `set()`), a new copy of the underlying array is created (hence the name).
- `CyclicBarrier` that provides a synchronization point (a barrier point) where a thread may need to wait until all other threads reach that point.

Self Test

1. Which of the following statements is true?
A. `ConcurrentNavigableMap` has two implementations, `ConcurrentSkipListMap` and `ConcurrentSkipListSet`
B. The `remove()` method of `CopyOnWriteArrayList` creates a new copy of the underlying array on which this class is based.
C. A `static` method can acquire the lock of an instance variable.
D. Constructors can be synchronized.

2. Which of the following options will correctly increment a value inside a map in a thread-safe way?

A.

```
Map<String, Integer> map = new ConcurrentHashMap<>();
int i = map.get(key);
map.put(key, ++i);
```

B.

```
ConcurrentMap<String, Integer> map = new ConcurrentHashMap<>();
map.put(key, map.get(key)+1);
```

C.

```
Map<String, Integer> map = new HashMap<>();
Map<String, Integer> map2 = Collections.synchronizedMap(map);
int i = map.get(key);
map.put(key, ++i);
```

D.

```
Map<String, AtomicInteger> map = new ConcurrentHashMap<>();
map.get(key).incrementAndGet();
```

3. Given:

```
public class Question_27_3 {
   public static void main(String[] args) throws Exception {
      BlockingDeque<Integer> deque =
            new LinkedBlockingDeque<>();
      deque.offerLast(10, 5, TimeUnit.SECONDS);
      System.out.print(
         deque.pollLast(5, TimeUnit.SECONDS)
         + " "
      );
      System.out.print(
         deque.pollFirst(5, TimeUnit.SECONDS));
   }
}
```

What is the result?
A. The program just prints `10`
B. The program prints `10` and hangs forever
C. After 5 seconds, the program prints `10` and after another 5 seconds, it prints `null`
D. The program prints `10` and 5 seconds later, it prints `null`

4. Under what circumstances can the `await()` method of `CyclicBarrier` throw an exception?
A. If the thread goes to sleep
B. When the last thread calls `await()`
C. If any thread is interrupted
D. If the number of threads that call `await()` is different than the number of threads `CyclicBarrier` was created with.

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

26. Thread Basics 28. Fork/Join Framework